Ball Balancing Along a Trajectory

Jochen Gast
TU Darmstadt

Björn Wild
TU Darmstadt

Abstract

Learning complex behavior in real robot scenarios is a challenging problem that requires solving several tasks simultaneously. Traditional reinforcement learning techniques, however, often favor experiments with a single challenge. Although such experiments are well suited for examining the performance and robustness of learning algorithms they may omit difficulties that may occur in combined tasks.

In this work, we address the problem of a combined task that requires learning both controller gains and a trajectory. Initially, we make use of a state-of-the-art policy search method to learn each behavior, respectively. Based on the results we finally learn a behavior for solving the combined challenge.

1 Introduction

In order to learn the combined task of balancing a ball on paddle along a trajectory we generally have to learn behavior for two different subtasks. First, we require the controller gains for balancing the ball and, second, the robot has to learn the parameters for the trajectory. Both tasks can be tackled as reinforcement learning problems that require trial-and-error episodes of the robot in order to iteratively improve its behavior. Policy search is an approach that has been successfully applied to various reinforcement learning problems with repeating episodes. However, standard methods such as stochastic finite-difference gradient (SFDG) often suffer from premature convergence. In [BS03] Bagnell and Schneider identified the problem being the loss of experience after non-covariant policy updates. Based on this inside Peters et al. developed a method that constraints the relative entropy of a robot's policy between consecutive policy updates [PMA10]. The resulting method *Relative Entropy Policy Search* (REPS) is sound, robust and, hence, suited for learning the controller gains and trajectory, respectively. Furthermore, REPS is able to handle different disturbances of the environment which makes it applicable for the combined task that is balancing a ball while the robot follows a trajectory.
For learning the trajectory we require an appropriate parametrization that allows the robot to learn plausible trajectories in joint space. To address this issue Kober et al. introduced Dynamic Movement Primitives (DMP), a template mechanism that encapsulates basic movement primitives in time dynamical systems [JK]. Based on both REPS and DMP we are finally able to solve both subtasks.

The rest of this report is divided into four parts. In section 2 we give a brief introduction to the computational framework of reinforcement learning including an algorithmic introduction to SFDG and REPS. Section 3 covers the evaluation of both methods in various experiments. We then present our solution to the combined task in section 4. Finally, in section 5 we give a conclusion and an outlook of what may be worth investigating in future.

2 From Reinforcement Learning to Policy Search Methods

Traditional reinforcement learning techniques as found in [SB98] treat reinforcement learning as a Markov decision process involving two interacting entities:

- An *agent* that constantly makes decisions about what *action* to perform next and
- an *environment* which is affected by the agent's actions and may subsequently change options and opportunities available for the agent.

Appearing in Proceedings of the 15^{th} International Conference on Artificial Intelligence and Statistics (AISTATS) 2012, La Palma, Canary Islands. Volume 22 of JMLR: W&CP 22.

In our scenario the agent is represented by a robot arm whereas the environment may be related to both the environmental disturbances and the robot's states such as joint configuration etc. The goal of the learning process is to figure out how the agent should optimally behave with respect to some feedback by the environment. The behavior of the agent is fully characterized by a *policy*

$$\mathbf{a} \sim \pi(\mathbf{a}|\mathbf{s}), \tag{1}$$

that maps states of the environment to actions. In the learning process the agent now tries to estimate which policy is best. Feedback of the environment is perceived by following functions:

- The *reward function* $r(\mathbf{s}, \mathbf{a})$ maps the agent's actions and the environment's states to an immediate reward which indicates how good the action u is when the environment is in state x.
- A *value function* $V(\mathbf{s})$ that maps a state to the accumulated reward an agent may expect when starting from state x.

Solving reinforcement learning problems is quite different from traditional well-defined solutions for supervised learning problems. Rather than having a set of training samples $D = \{\mathbf{s}, \mathbf{a}\}_i$ available which possibly allows the learner to infer the intrinsic structure of the problem, only a reward signal $r(\mathbf{s}, \mathbf{a})$ is available for the learner. Indeed, it is the responsibility of the learner himself to explore which actions give the most reward by trying them out. There are basically two approaches to tackle reinforcement learning problems, value function methods and policy search.

Value function methods have historically been the driving force of reinforcement learning in the 1990s [Gor95, Bai95]. After iteratively computing the value function $V(\mathbf{s})$ for each state, the policy is obtained either as a final step (*value iteration*) or is consecutively improved in each iteration (*Policy Iteration*). As this only works for a limited class of systems, approximations exist, e.g. in the form of *Q-Learning* [Wat89]. Unfortunately, value function methods have the drawback of being reliable on filling up the state-action space. This makes it hard to solve high-dimensional problems such as robot-learning tasks.

Policy search, on the other hand, represents a more recent approach which tries to estimate the policy directly. Its proceeding is shown in table 1:
Given a update strategy $f(\mathbf{r}, \mathbf{s}, \mathbf{s}')$ we initialize our policy with π_0 and start to improve our policy iteratively. In each iteration the current policy π_k is evaluated within the system. Based on rewards, states and corresponding following states a new improved policy is then computed in the update step. In contract to value function methods policy search is able to make use of expert's knowledge. To begin with we may start learning with an initial policy based on domain knowledge. As a result of that policy search does not necessarily have to explore the whole state-action space but just the local area around the expert's suggested policy.

Input: update strategy $f(\boldsymbol{r}, \boldsymbol{s}, \boldsymbol{s}')$
Initialization: $\pi_0 = \pi_0(\boldsymbol{s}\|\boldsymbol{a}, \boldsymbol{\theta})$
For each iteration k
Evaluation: $(r_i, \boldsymbol{s}'_i) \leftarrow \pi_k(\boldsymbol{s}\|\boldsymbol{a}, \boldsymbol{\theta})$
Update: $\boldsymbol{\theta}_{k+1} = f(r_i, \boldsymbol{s}_i, \boldsymbol{s}'_i)$
Output: π_{final}

Table 1: Policy Search. We start exploration from an initial policy π_0. Based on rewards r_i, states $\mathbf{s}_i$ and corresponding following states $\mathbf{s}'_i$ we iteratively improve the policy with the update strategy f.

In the horizon of this work we consider two rather different approaches to compute policy updates introduced in sections 2.1 and 2.2.

2.1 Stochastic Finite Difference Gradient

The SFDG approach computes a policy update by approximating $\nabla_\theta J$, the gradient of the reward w.r.t. to the policy. The policy update rule then is

$$\theta_{k+1} = \theta_k + \alpha \nabla_\theta J, \tag{2}$$

where the learning rate α limits change allowed in consecutive iterations. In order to approximate the gradient we utilize the first order Taylor approximation

$$J(\theta + \delta\theta) = J(\theta) + \delta\theta^T \nabla_\theta J \tag{3}$$

$$\Leftrightarrow J(\theta + \delta\theta) - J(\theta) = \delta\theta^T \nabla_\theta J. \tag{4}$$

For given samples we can use linear regression to approximate the gradient as

$$\nabla_\theta J = (\Theta^T \Theta)^{-1} \Theta^T \mathbf{b}, \tag{5}$$

$$\Theta = \begin{pmatrix} \delta\theta_1^T \\ \delta\theta_2^T \\ \vdots \\ \delta\theta_N^T \end{pmatrix} \qquad \mathbf{b} = \begin{pmatrix} \delta J_{\theta_1} \\ \delta J_{\theta_2} \\ \vdots \\ \delta J_{\theta_N} \end{pmatrix}, \tag{6}$$

where $\delta\theta_i$ denote roll-outs around the current policy θ and $\delta J_{\theta_i} = J(\theta + \delta\theta_i) - J(\theta)$ their corresponding differences of received reward. Note that it is common to normalize the gradient afterwards to assure improvement steps to have the same magnitude.

2.2 Relative Entropy Policy Search

One drawback of classical policy search methods such as SFDG is premature convergence. The problem here is the policy update step. That is an agent taking policy updates only in the direction of steepest ascent discards knowledge from exploration made in previous iterations. The consequence is a *loss of information* that could have helped the agent in the most recent update step. REPS addresses this issue by constraining the information loss between consecutive iterations where the agent itself draws actions from a stochastic policy $\pi(\mathbf{a}|\mathbf{s})$ which we represent by a linear Gaussian model $\sim N(\mathbf{b} + \mathbf{s}F|\Sigma)$. The observed data distribution as produced by current policy is denoted by $q(\mathbf{s}, \mathbf{a})$ while the joint distribution of the new policy is given by $p(\mathbf{s}, \mathbf{a})$. The difference between these distributions can be represented by the Kullback-Leibler divergence $KL(p||q)$ and the information loss can be bounded by

$$KL(p||q) = \sum_{\mathbf{s},\mathbf{a}} p(\mathbf{s}, \mathbf{a}) \log \frac{p(\mathbf{s}, \mathbf{a})}{q(\mathbf{s}, \mathbf{a})} \leq \epsilon, \tag{7}$$

where ϵ is the maximum loss of information.
REPS now obtains a new policy by solving the problem:

$$\max_{\pi} J(\pi) = \sum_{\mathbf{s},\mathbf{a}} p(\mathbf{s}, \mathbf{a}) r(\mathbf{s}, \mathbf{a}) \tag{8}$$

$$\epsilon \geq \sum_{\mathbf{s},\mathbf{a}} p(\mathbf{s}, \mathbf{a}) \log \frac{p(\mathbf{s}, \mathbf{a})}{q(\mathbf{s}, \mathbf{a})} \tag{9}$$

$$\sum_{\mathbf{s},\mathbf{a}} p(\mathbf{s}, \mathbf{a}) \phi(\mathbf{s}) = \sum_{\mathbf{s},\mathbf{a}} q(\mathbf{s}, \mathbf{a}) \phi(\mathbf{s}) = \hat{\phi} \tag{10}$$

$$1 = \sum_{\mathbf{s},\mathbf{a}} p(\mathbf{s}, \mathbf{a}), \tag{11}$$

where the goal is to maximize the expected reward (8) under the constrained information loss (9). Equation (10) assures that the policy will be able to deal with various states. More precisely, we assure the new distribution $p(\mathbf{s}, \mathbf{a})$ not only to match state-action pairs with high rewarded states but also pairs with less promising states by making the expected features be compliant with the observed ones.

A solution to this optimization problem can be obtained using Lagrangian multipliers and is given by

$$p(\mathbf{s}, \mathbf{a}) = q(\mathbf{s}, \mathbf{a}) \exp\left(\frac{r(\mathbf{s}, \mathbf{a}) - V(\mathbf{s})}{\eta}\right) Z^{-1} \tag{12}$$

$$Z = \sum_{\mathbf{s},\mathbf{a}} \exp\left(\frac{r(\mathbf{s}, \mathbf{a}) - V(\mathbf{s})}{\eta}\right), \tag{13}$$

where $V(\mathbf{s}) = \theta^T \phi(\mathbf{s})$ denotes the value function which can be obtained by minimizing the dual function

$$\min_{\eta,\theta} g(\eta, \theta) = \eta \log(Z') + \eta\epsilon + \theta^T \hat{\phi} \tag{14}$$

$$Z' = \sum_{\mathbf{s},\mathbf{a}} q(\mathbf{s}, \mathbf{a}) \exp\left(\frac{r(\mathbf{s}, \mathbf{a}) - V(\mathbf{s})}{\eta}\right). \tag{15}$$

The overall algorithm is given in table 2. After having obtaining η and θ we compute the new policy with a maximum likelihood estimate based on the samples.

Input: max information loss ϵ, feature transform $\phi(\boldsymbol{s})$
Initialize: $\pi = \pi_0(\boldsymbol{a}|\boldsymbol{s})$
For each iteration k
Generate Samples: $(r_i, \boldsymbol{s}_i, \boldsymbol{a}_i)$ for $i = 1...N$
Obtain Features: $\phi_i = \phi(\boldsymbol{s})$ for $i = 1...N$
Minimize dual function: $\min_{\eta,\boldsymbol{\theta}} \; g(\eta, \boldsymbol{\theta}) = \eta \log(Z') + \eta\epsilon + \boldsymbol{\theta}^T \hat{\boldsymbol{\phi}}$ $Z' = \sum_{\boldsymbol{s},\boldsymbol{a}} q(\boldsymbol{s}, \boldsymbol{a}) \exp(\frac{r(\boldsymbol{s},\boldsymbol{a}) - V(\boldsymbol{s})}{\eta})$
Policy update: Calculate $\pi_k(a|s)$ by Maximum-Likelihood estimation
Output: policy $\pi(a|s)$

Table 2: REPS

3 Experiments

In this section we cover the evaluation of various experiments. First, we will test REPS on a reward function containing states using different parameter settings. Finally, SFDG and REPS are compared using both an state-less reward function and a physically-based simulation.

3.1 REPS with different parameter settings

Various parameter settings of REPS are evaluated on a reward function given by

$$r(\mathbf{s}, \mathbf{a}) \sim N(\mu_\mathbf{r}, \Sigma_\mathbf{r}), \quad \mu_\mathbf{r} = (1\ 2\ 3\ 4\ 5)^T, \quad \Sigma_\mathbf{r} = 5\,\mathbb{E},$$

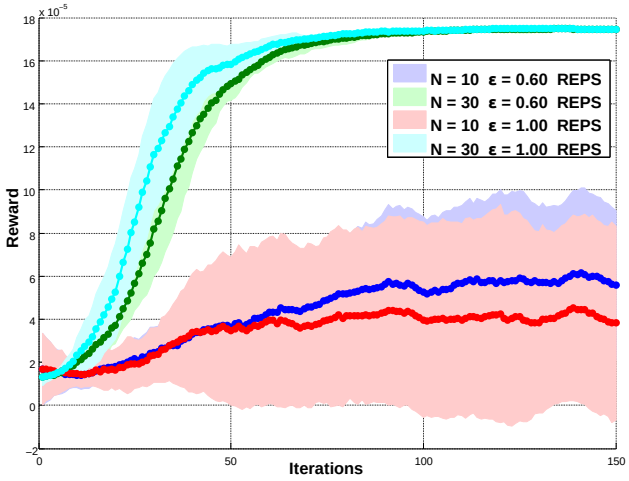

Figure 1: REPS on evaluating a state-action space of dimension 5. Note that $\epsilon = 1.0$ allows larger steps than $\epsilon = 0.6$. However, given fewer samples both algorithms fail to converge within 150 iterations.

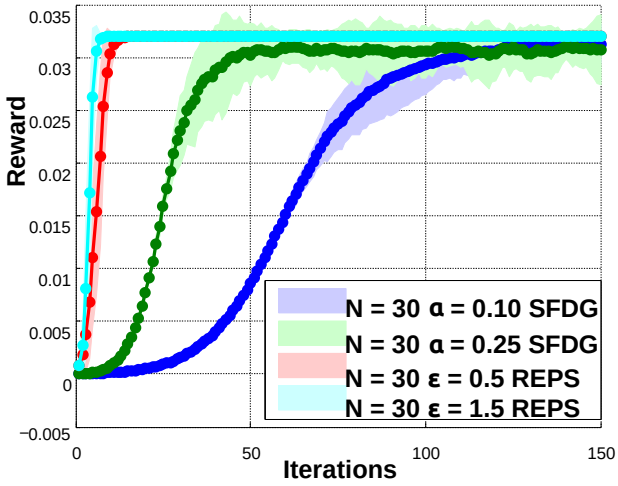

Figure 2: Comparison of SFDG vs REPS in a two dimensional state-space. Both algorithms were 30 times for 70 iterations. Both REPS parameter settings have an edge on the corresponding SFDG iterations.

where the state dimension is $|\mathbf{s}| = 3$ and the action dimension is $|\mathbf{a}| = 2$. Reward for a given state-action pair then is evaluated at $N(\mathbf{x}|\mu_\mathbf{r}, \Sigma_\mathbf{r})$ by concatenating them to $\mathbf{x} = (\mathbf{s}^T\ \mathbf{a}^T)^T$. Furthermore, we generate initial states for the problem by drawing samples $\mathbf{s}_i$ from

$$\mathbf{s} \sim N(\mu_\mathbf{s}, \Sigma_\mathbf{s}), \quad \mu_\mathbf{s} = (0\ 0\ 0)^T, \quad \Sigma_\mathbf{s} = 0.05\,\mathbb{E}.$$

Exploration is initially started from:

$$\pi_0 = (0\ 0\ 0\ 0\ 0)^T, \quad \Sigma_0 = 30\,\mathbb{E}$$

that is, not having prior knowledge in the first place we allow a large variance of generated actions. Figure 1 shows the resulting rewards for 150 iterations where the bounds $\{\epsilon = 0.6, \epsilon = 1.5\}$ and sample sizes $\{N = 10, N = 30\}$ were the configurations. Following observations can be made:

- For $\epsilon = 0.6$ and $\epsilon = 1.5$, respectively, having more samples available improves the convergence rate. This is due to the fact that more samples represent the state-action space more accurately. However, having a higher sampling size may be expensive when solving a optimization problem in a simulator or even on a real robot.
- Allowing a higher change in relative entropy between consecutive iterations gives a better convergence rate. This is not necessarily true in general as we state in the original motivation of using REPS. By allowing the policy differ too much from the current one, we actually loose information which we accumulated up to the current iteration. Note that convergence actually breaks down when $\epsilon = 1.5$ as compared to $\epsilon = 1.0$.
- In general we noticed that using ϵ 1.0 gives good performance in most scenarios.

3.2 SFDG vs REPS

State-less Reward Function

We evaluated the performance of SFDG vs REPS in a two dimensional state- action space where we chose

$$r(s,a) \sim N(\mu, \Sigma) \quad \mu = (5,3)^T \quad \Sigma = \begin{pmatrix} 5 & 0.5 \\ 0.5 & 5 \end{pmatrix}$$

to be a Gaussian reward function that simply joins states and actions. Both SFDG and REPS initially start their exploration from $\pi_0 = (0\ 0)^T$. Again, the initial variance of the policy is set to a high variance

$$\Sigma_0 = \begin{pmatrix} 10 & 0 \\ 0 & 10 \end{pmatrix} \tag{16}$$

in order to allow a wide exploration. Note that SFDG will use the same variance to compute its approximation of the gradient. Figure 2 shows the results of running both algorithms 30 times for 70 iterations, respectively. Thereby, the mean reward is plotted as well as a tube consisting of two standard deviations around the mean. The sampling size was set to $N = 30$ and both algorithms were run with two different parameter settings. That is SFDG was run with $\alpha = 1$ and $\alpha = 0.25$ and REPS was run with $\epsilon = 0.1$ and $\epsilon = 1.0$. Following observations can be made:

- REPS outperforms SFDG both with $\epsilon = 0.5$ and $\epsilon = 1.5$ right from the beginning.

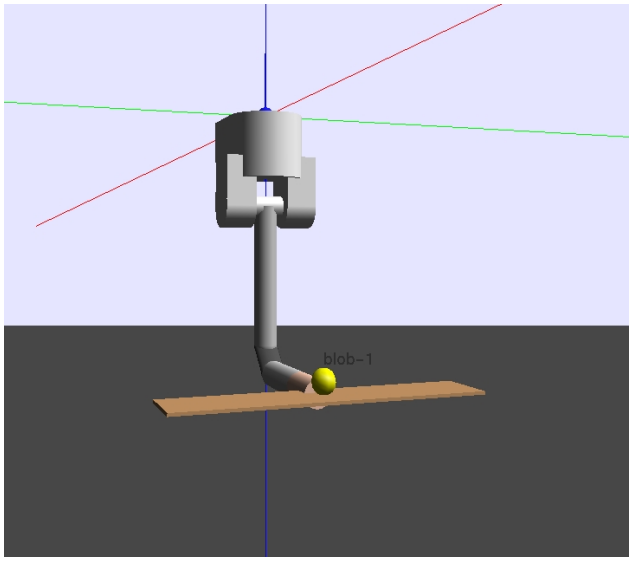

Figure 3: The Ball on a Beam task.

- REPS with a higher bound $\epsilon = 1.5$ on the information loss converges the fastest.By allowing the algorithm to make bigger changes in relative entropy consecutive policy updates make give larger improvements. Note that the difference between $\epsilon = 0.5$ and $\epsilon = 1.5$ consist only during the first 15 iterations.

- The variance of runs with $\epsilon = 0.5$ is higher than the variance for runs with $\epsilon = 1.5$. This is due to the fact that restricting the policy updates, also restricts its change from the starting policy which has a comparatively high variance Σ_0. The magnitude then is taken over until convergence.

- During the first 100 iterations SFDG with $\alpha = 0.25$ has a big advantage over SFDG with $\alpha = 0.1$ but oscillates around the solution.

Ball on a Beam

Furthermore, we tested SFDG vs REPS in a physically-based environment where we used both approaches to optimize the policy for balancing a ball on a beam as shown in figure 3. Thereby, the learners have to obtain two controller gains for one dimension. The ball's initial velocity in the length direction of the beam was set to $v_{x,0} = 0.5$.

Both learning algorithms were run for 20 iterations with $N = 30$ samples. While SFDG was trained with $\alpha = 0.25$, REPS was set up with $\epsilon = 0.6$. We set the initial policy to

$$\pi = (0 \quad 0)^T, \qquad \Sigma_0 = \begin{pmatrix} 0.1 & 0 \\ 0 & 0.1 \end{pmatrix} \tag{17}$$

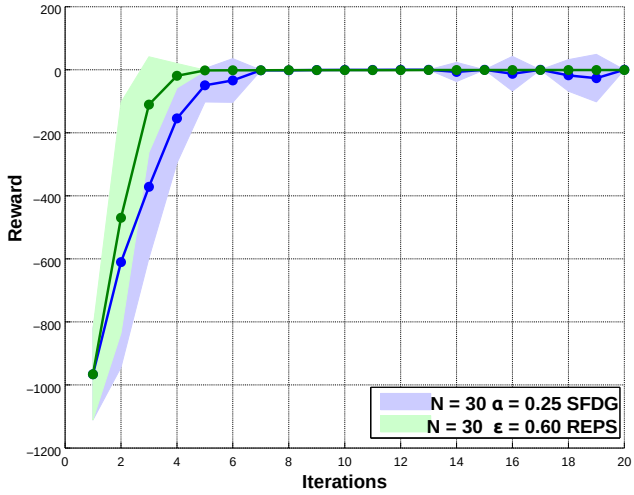

Figure 4: The resulting rewards for the Ball on a Beam task.

including our domain knowledge that the required gains are rather small. The resulting rewards over iterations are shown in figure 4. We observe:

- REPS convergences faster than SFDG, however, the difference is marginal.

- When SFDG oscillates a couple of times after reaching local optimum. This is no surprise as it continues to approximate the gradient from its found optimum.

- REPS, on the other hand, still continues to improve its policy after being close to convergence. These improvements are rather small, however, subtle improvements in reward are still interesting when trying to find the best possible policy.

4 Ball Balancing Along a Trajectory

The following sections cover our solution to the combined task. At first, we discuss balancing a ball on a paddle. Independent from the first step, we then learn a trajectory that moves through two given points in task space. Finally, we use the results of both subtasks to develop a learning process for the combined task.

4.1 Balancing

Although the balancing task is similar to the ball on a beam experiment in section 3.2 we have to make a few adjustments in order to make it applicable as an initial policy for the combined task. In contrast to ball on a beam where we restricted the problem to one dimension we now face both a two dimensional

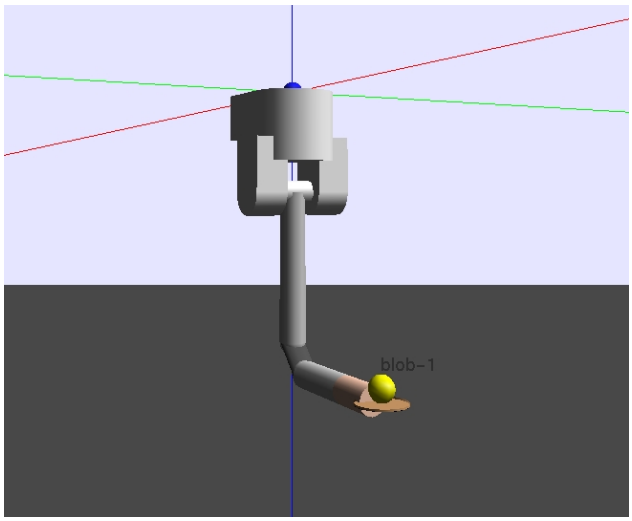

Figure 5: Balancing a ball on a paddle.

problem and a smaller area where we want to balance the ball in (as shown in figure 5). Furthermore, states are used while learning as the robot should be able to to cope with different initial situations. We selected the ball velocity $\mathbf{s} = (\mathbf{v}_{\text{ball},x}\mathbf{v}_{\text{ball},y})^T$ to be the states our robot should adapt to.

The reward is evaluated for a given state and an action in one episode by

$$r(\mathbf{s},\mathbf{a}) = -\sum_{t=0}^{T}(\,0.1||\mathbf{x}_{\text{ball},t} - \mathbf{x}_{\text{paddle},t}||^2 - \sum_j ||\ddot{q}_{t,j}||^2), \tag{18}$$

where T denotes the length of an episode, $||\mathbf{x}_{\text{ball},t} - \mathbf{x}_{\text{paddle},t}||^2$ is the L_2-norm of the relative difference of the ball's and the paddle's position and $\ddot{q}_j$ are the robot's joints. While the first term assures the ball to be pushed back into the middle, the latter term desires small joint accelerations. In other words fast movements of the robots are punished. It turns out that the task can be learned sufficiently well in about 150 iteration as shown in figure 6. Note that the resulting gains are suitable for low disturbances of the ball, while the robot fails to balance the ball when we use higher initial velocities. However, this is not surprising since for high velocities a human is only able to balance a table tennis on a paddle after a long time of training. In the end we do not prematurely optimize the resulting gains of this subtask as they only serve as a starting point for the combined task.

4.2 Trajectory

As mentioned in the introduction we make use of dynamic movement primitives (DMP) to learn a trajectory. Detailed descriptions of DMP can be found in [JK, DNP12, DNKP13] yet a detailed explanation of DMP is not within the horizon of this work. Generally, DMP allows us to parameterize a generic trajectory in the joint space through radial basis functions. For our task such a parametrization requires

$$\mathbf{q}_0 = \mathbf{q}(t_0), \tag{19}$$
$$\dot{\mathbf{q}}_0 = \dot{\mathbf{q}}(t_0), \tag{20}$$
$$\mathbf{q}_T = \mathbf{q}(t_T), \tag{21}$$
$$\mathbf{w}_i = \mathbf{w}_{i,const}, \qquad i = 1...N_{\text{RBF}} \tag{22}$$

where $\mathbf{q}_0$, $\dot{\mathbf{q}}_0$ is the initial joint configuration and joint velocity, respectively, $\mathbf{q}_T$ is the final joint configuration, T is the length of the episode, $\mathbf{w}_i$ are weights that influence the width of the radial basis functions and N_{RBF} is the number of radial basis functions. The weights $\mathbf{w}_i$ correspond to parameters θ_i that are obtained from learning a DMP. We set

$$T = 4000 \tag{23}$$
$$N_{\text{RBF}} = 4 \tag{24}$$
$$\mathbf{q}_0 = (0\,0\,0\,\frac{\pi}{2}) \tag{25}$$
$$\dot{\mathbf{q}}_0 = (0\,0\,0\,0] \tag{26}$$
$$\mathbf{q}_T = (0\,0\,0\,\frac{\pi}{2}), \tag{27}$$

which means our movement will start from an initial position with zero velocity and return to that position again at $t = T$. Note that we set the remaining three joints to zero for all time steps as the controller should adapt them to the ball state when executing the combined task. Furthermore, two via points are placed in task space at

$$\mathbf{x}_{\text{ball}_1} = (-0.2 \quad -0.65 \quad -0.85)^T, \tag{28}$$
$$\mathbf{x}_{\text{ball}_2} = (\ 0.3 \quad -0.65 \quad -0.7)^T. \tag{29}$$

as shown in figure 10.

In order to learn the trajectory we use both REPS and DMP where we set the reward function to

$$r(a) = \sum_j ||\ddot{q}_{t,j}||^2 - \sum_{i=1}^{2} \min_{t\in[0,T]} ||\mathbf{x}_{\text{cart},t} - \mathbf{x}_{i,t}||^2. \tag{30}$$

Thus, we penalize high accelerations and sum up the squared difference of the minimum distance between the two via points and the end-effector. As shown in figure 6 REPS is able to learn the desired trajectory within 150 iterations. The resulting trajectories of the joints are depicted in figure 7

4.3 Balance a Ball on a Paddle Along a Trajectory

For the combined task we require the results of the previous sections. That is the agent moves along the

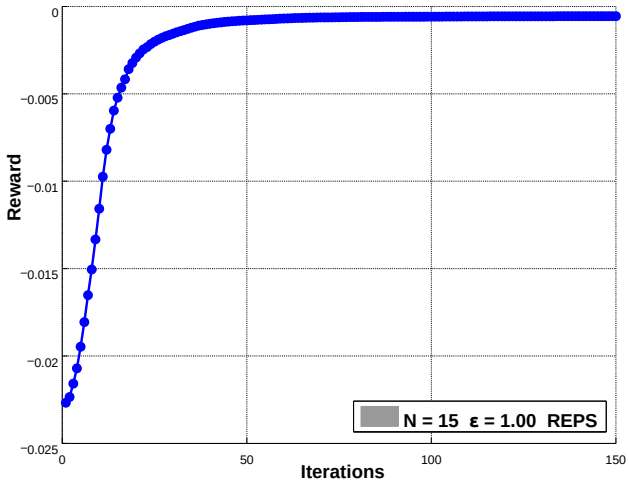

Figure 6: Development of the average rewards while learning the trajectory.

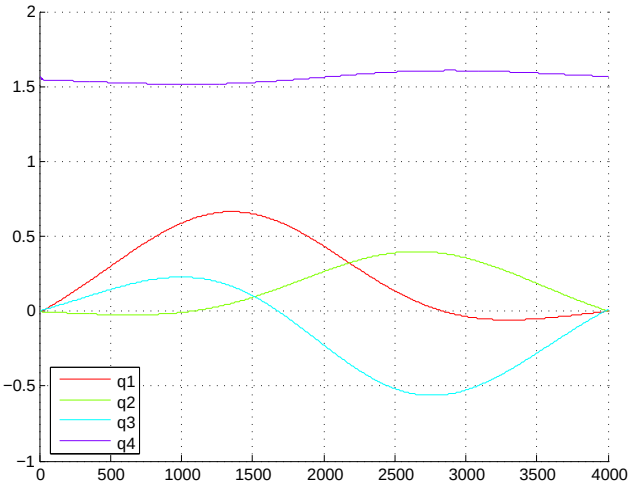

Figure 7: Smooth joint trajectories as generated by REPS + DMP.

trajectory as learned in section 4.2. The robot now has to keep track of the ball so that it does not drop which can be achieved by using a controller at the same time. We identify following issues:

- As far as control is concerned the problem differs from the standard balancing task quite a bit. While we have an artificial known state distribution during the learning process of the balancing task, we are faced with unknown states that possibly change over time. A solution may consist in learning a trajectory for the controller gains, however, we explain a different approach in the following sections.
- Hence, it is not trivial to identify an initial state distribution. For this reason we have to design a way receive the actual state distribution from the robot's simulation.
- Since the paddle is now moving around in task space, we include the ball's relative position into the state. Additionally, the absolute velocity difference between the ball and the paddle has to be changed to relative velocity difference.
- Furthermore, we have to rotate the states w.r.t. the end-effector's current orientation.

Our solution consists of two key concepts which we explain in the following sections. First, we develop a way to make our agent adapt the actual state distribution. And second, we present a way to overcome the issue that the state- action space is considerably harder to explore than in the previous tasks.

Adapting the State Distribution

As mentioned before one problem of the combined task is to estimate the actual state distribution. The goal here is to estimate a good linear transformation matrix F that results in robust performance for the whole trajectory. To approximate the actual distribution we apply following technique as depicted in table 3 and 8. The REPS interface for the proposed solution is shown in 3. The key idea is the following: Having defined a frequency $\Delta t_{\text{observe}}$ we reset the actions given the constant linear transformation F every $\Delta t_{\text{observe}}$ time steps. For this reason state-less action samples are drawn for each episode by setting the state $\mathbf{s} = 0$. The simulator receives these actions together with the linear transformation F and the observation frequency $\Delta t_{\text{observe}}$. As a result the learning algorithm obtains states $\mathbf{s}_0, \mathbf{s}_1, ...\mathbf{s}_M$ and the reward for the whole episode. In order to reconstruct the applied actions within the simulator we have to apply the linear model again to finally obtain samples $(\mathbf{s}_i, \mathbf{a}_i, r_i)$.

REPS

For each iteration:

Initialize:
S = ∅, A = ∅, R = ∅

For each episode:

Draw state-less action: $\boldsymbol{b} = \pi(\boldsymbol{a}|\boldsymbol{s} = 0)$

Simulation:
$(\boldsymbol{s}_0, \boldsymbol{s}_1, ..., \boldsymbol{s}_M, r) \leftarrow \text{runEpisode}(\boldsymbol{b}, F, \Delta t_{\text{observe}})$

Adapt to actual State distribution:

$$\begin{array}{ll} (\boldsymbol{s}_0, \boldsymbol{a}_0, r) & \boldsymbol{a}_0 = \boldsymbol{b} + \boldsymbol{s}_0 F \\ (\boldsymbol{s}_1, \boldsymbol{a}_1, r) & \boldsymbol{a}_1 = \boldsymbol{b} + \boldsymbol{s}_1 F \\ (\boldsymbol{s}_2, \boldsymbol{a}_2, r) & \boldsymbol{a}_2 = \boldsymbol{b} + \boldsymbol{s}_2 F \\ \vdots & \vdots \\ (\boldsymbol{s}_M, \boldsymbol{a}_M, r) & \boldsymbol{a}_M = \boldsymbol{b} + \boldsymbol{s}_M F \end{array}$$

Accumulate samples:

$$\begin{array}{ll} S = S \cup \boldsymbol{s}_i & i = 1...N \\ A = A \cup \boldsymbol{a}_i & i = 1...N \\ R = R \cup r_i & i = 1...N \end{array}$$

Minimize Dual Function

$\vdots$

Table 3: REPS Part.

SIMULATOR

runEpisode($\boldsymbol{b}, F, \Delta t_{\text{observe}}$):

Initialize: S = ∅, $r = 0$

For each timestep t:

Simulate Physics: $\boldsymbol{s}_t = \text{simulateTimestep}(t)$

if (mod(t, $\Delta t_{\text{observe}}$) == 0)

$\boldsymbol{a} = \boldsymbol{b} + \boldsymbol{s}_t F$

$S = S \cup \boldsymbol{s}_t$

Accumulate Reward: $r = r + \text{reward}(\boldsymbol{a}, \boldsymbol{s}_t)$

Output: S, r

Figure 8: SL Part.

That is we associate the reward of an episode together with all states and actions that occurred in the episode with the observation frequency. For each iteration the obtained states, actions and rewards are accumulated in sets (S, A, R) as these are required in the next optimization step. Setting the observation frequency is not trivial. A high frequency results in a lot of samples which then are used in the optimization step. Hence, computational performance may slow down. However, setting a low frequency may not give accurate results of the state distribution over the whole episode.

The interface call to the simulator is shown in table 8. The episode is run with the given state-less action $\mathbf{b}$, linear transformation matrix F and the observation frequency $\Delta t_{\text{observe}}$. For consecutive time steps the actual actions $\mathbf{a}$ are then updated with the observation frequency. While the reward is accumulated for all time steps, states $\mathbf{s}_t$ as only used in the update step are remembered in a set S. Finally, all states and the reward are returned to the REPS interface.

Coarse-to-Fine Estimation

In order to overcome the hard exploration of the state-action space we exploit the ability of policy search methods to easily integrate expert knowledge. However, since we have no expert knowledge available we have to create some knowledge first. In order to gather insights on the problem we apply following trick: Instead of learning the desired actions from actions on the small paddle, we increase the paddle size to learn the combined on a big one. As a result of that learning will be easier and hence more robust. We then initialize the policy for learning on a smaller paddle with the obtained policy. In our case it turns out that the controller gains as obtained of a paddle with double the size are a reasonable initialization for learning on the original paddle. Note that the learning process for the coarse paddle was initialized with gains as obtained in the first subtask 4.1.

Reward Function

The general form of the selected reward function is given by

$$\begin{aligned} r(\mathbf{s}, \mathbf{a}) = &- \alpha \, \text{penalty}_{\text{stepover}} \\ &- \beta \, \text{penalty}_{\text{ddQ}} \\ &- \gamma \, \text{penalty}_{\delta \mathbf{x}} \\ &- \delta \, \text{penalty}_{\delta \mathbf{v}}, \end{aligned} \tag{31}$$

where $\text{penalty}_{\text{stepover}}$ penalizes whenever the ball drops, $\text{penalty}_{\text{ddQ}}$ desires low accelerations, $\text{penalty}_{\delta \mathbf{x}/\delta \mathbf{v}}$ penalizes the relative difference of the

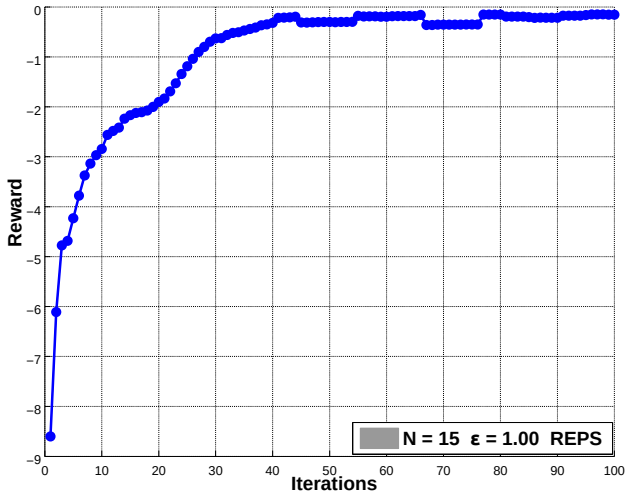

Figure 9: Learning process for the combined task of balancing a ball on a paddle.

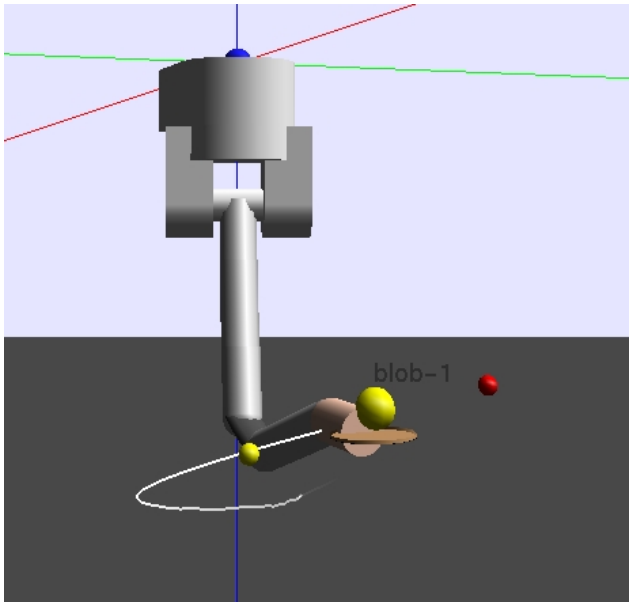

Figure 10: Balancing a Ball Along a Trajectory. While the trajectory is learned in a independent step, the controller gains are updated as the robot moves along the trajectory.

ball's and the paddle's position/velocity, respectively. The resulting behavior is depicted in figure 10 and the resulting learning process is shown in 9. With the help of coarse to fine estimation the robot is able to start exploration from a good initial policy. However, note that the ball is not centered during the whole trajectory but it moves around the center depending on the current velocity of the paddle.

5 Conclusion & Outlook

In this section give a brief overview over issues we noticed and a final outlook All in all, we made following final observations:

- When implementing the optimization algorithms one has to pay attention to numerical stability. Especially, when very slow numbers are involved numerical stability becomes an issue.
- Bootstrapping is a useful technique when training agents on various tasks. Reusing a resulting policy of a previous learning process saves a lot of time.
- Finding the right reward function is not trivial. Even after careful optimizing one may make a few adjustments to get even a more robust policy.
- There is a trade-off between learning a specific task very well and learning it sufficiently well with the ability to generalize well. For instance, in the combined balancing-trajectory task we may find a policy that keeps the ball centered, however, solutions that allow the ball to move around the center a more robust to disturbances.

In future we may investigate the generalization ability of the provided solution. Once the linear transformation matrix for the states is learned, we may actually be able to apply the same matrix in a setup with a completely different trajectory.

References

[Bai95] Leemon Baird. Residual algorithms: Reinforcement learning with function approximation. In *In Proceedings of the Twelfth International Conference on Machine Learning*, pages 30–37. Morgan Kaufmann, 1995.

[BS03] J. Andrew Bagnell and Jeff Schneider. Covariant policy search, 2003.

[DNKP13] Christian Daniel, Gerhard Neumann, Oliver Kroemer, and Jan Peters. Learning sequential motor tasks, 2013.

[DNP12] Christian Daniel, Gerhard Neumann, and Jan Peters. Learning concurrent motor skills in versatile solution spaces, 2012.

[Gor95] Geoffrey J. Gordon. Stable function approximation in dynamic programming. In *IN MACHINE LEARNING: PROCEEDINGS OF THE TWELFTH INTERNATIONAL CONFERENCE*. Morgan Kaufmann, 1995.

[JK] Christoph H. Lampert Bernhard Schölkopf Jan Peters Jens Kober, Katharina Mülling Oliver Kroemer. Movement templates for learning of hitting and batting.

[PMA10] Jan Peters, Katharina Mülling, and Yasemin Altün. Relative entropy policy search, 2010.

[SB98] Richard S. Sutton and Andrew G. Barto. Reinforcement learning: An introduction, 1998.

[Wat89] Christopher J. C. H. Watkins. *Learning from Delayed Rewards*. PhD thesis, King's College, Cambridge, UK, 1989.